(BI 66/1/18

MTB 373

TO: MTB Distribution

FROM: Bernard S. Greenberg

Date: 04/18/78

Subject: Real-time Editing on Multics

I. What is it?

Real-time editing is the ability to edit or enter text seeing at all times what one has entered, as it sits in your segment, not as you have typed it. Real-time editing is the ability to modify a text, watching what happens as each command is typed, as opposed to asking the editor "Now, let's see what you did for what I told you". Real-time editing is taking advantage of intelligent and semi-intelligent display terminals: utilizing their ability to display text and alter it without retransmitting the entire screen. Real-time editing is the ability to say "Change this 9 HERE to a 10" as opposed to "Change the 9 which is followed by a close parenthesis, a space, a 17, a comma and a 0 to a 10", and find out that there were two of them.

Real-time editing brings the ability to have the text-entry subsystem participate in the preparation of text. Language assists, such as automatic PL/I indentation, "shorthand" expansion, or text justification, can be performed as the actual text is being typed in. Need to indent, expand, or post-process to match "begin"'s, "end"'s, or parentheses, vanishes.

Real-time editing is the text-processing technology of the 1980's, based on the latest data terminal technology of today. We can have this now.

II. Why do we want it?

Text preparation and program editing time are reduced by orders of magnitude by real-time editing. Instead of encoding instructions to a conventional text editor to try to convince it to do what you want, you DO what you want, watching what you are doing as you are doing it. Almost all of the steps of conventional editing vanish. Most notable are the absence of the "print" command and the substitution string, the most common of editor inputs.

A service-quality printing terminal (e.g., Terminet 300) can be purchased for $3000, and can communicate at 300 baud. A 1200 baud terminet is around twice that price. Printing terminals are noisy, slow, expensive, and difficult to maintain. They generate waste paper, and are in and of themselves a constraint on communications technology.

Multics Project internal working documentation. Not to be reproduced or distributed outside the Multics Project.

Microprocessor technology drives the price of intelligent terminals down daily. Intelligent display terminals capable of operating at 19.2 KILOBAUD are available for $800 or so. Even an extremely clever display terminal (i.e., the Delta Data 4000) is cheaper than the 300 baud terminet. Several Multics sites rely heavily on display terminals; they are the data entry technology of the present. The microprocessor will consign the printing terminal to the same place it consigned the slide rule.

Multics is now very weak in its support of display terminals. Users of such terminals, whose numbers are increasing in the Multics community, can now choose between "page length" processing, or watching data float off the screen at line speed. On the input side, editing using a display terminal on Multics is particularly frustrating, for text being edited vanishes off the screen as fast as you can type editing commands, and there is no hard copy to look back at. Indeed, most time spent attempting to use "qedx" or "edm", or even "TECO" from a display terminal is spent saying, "Let's look at these few lines again and see what they say". This is absurd, for video terminal technology was designed to facilitate editing of data as you see it.

Multics must adapt to video terminal technology, and support the growing variety of video terminals in the same flexible and general way it has supported printing terminals. Multics must keep in step with the forefront of technology from which it came. Multics must have the best, and be the best.

III. How does it work (on the surface)?

Real-time editing is accomplished by invoking an editor, a user-ring Multics program, as are conventional editors. The editor clears the terminal screen, except for a line near the bottom stating constant information, such as the fact that the editor is being conversed with, a "buffer name", and the pathname of what is being edited, if any.

At once, the user can begin to type text. There is no "input mode" or "edit mode". As the user types text, including newline, space, and tab characters, the characters appear on the screen as he or she types them. If a mistake is made, the striking of a "#" causes the previous character to be wiped off the screen as though it were never typed, as an @ will strike out a line. Even at this elementary level, the text appears correct and as intended at all times, instead of conglomerations of pound signs and at-signs. The displayed text is always right. The column position is always accurate (unlike today, when line editing forces us to type
"\fa" if we care what column we are at).

As the user types text, it is entered into a "buffer," as in conventional editors, which may have been initially filled from a segment, and may be written out to that or some other segment. As characters are entered, the terminal's cursor (a blinking mark which "runs" (lat., "curreo", I run) all over the screen) provides a visible indication where the next character will go. Normally, it is in the position right after the last character typed, so characters follow each other as they are typed. The basic notion of real-time editing identifies the cursor, on the screen, with a conceptual "pointer" into the editor's buffer, such as the "current line" of qedx or edm, or even more closely, the "." of TECO. The cursor is maintained by the editor to be at that character on the screen which is the "current character of the current line" in the editor's conceptualization. Thus, normal entry of characters "moves" the editor's "pointer" as characters are typed, which, as a side effect, moves the cursor on the screen as well.

Each character typed in this way may thus be viewed as a command to the editor to "insert" that character at the current "pointer", and move the pointer to the right of that character. Thus, typing "a", "b", and "c", in sequence means "put in an "a", a "b", and a "c"". Contrast this with qedx, where "a" means "append", "b" means "go to buffer", and "c" means "change". Similarly, "#" means "make the character to the left of the "pointer" disappear", which is what it has always meant in Multics anyway, but in the context of the real-time editor, the character actually disappears: both from the buffer and the screen.

Now the editor can be convinced to move its "pointer" around to any text in the entire buffer. This might be useful, for example, to insert a word between two words we had already typed. If we could convince the editor to move the "pointer" between the two words, simply typing the new word would put those characters right there (for we have already decided that "a" meant put an "a" to the right of the pointer and move the latter by one). As they would be typed, the user would see them appear in the middle of the chosen line as he or she typed them, the rest of the line moving over to the right as the new characters were typed, one by one. And if a mistake were made, just typing "#" would delete the character to the left of the pointer, as it always does, and that character would summarily disappear from the screen and the buffer, the rest of the line moving to the left visibly.

The editor can be convinced to move its pointer by typing "non-printing characters" at it. For instance, "control B"

(1) move the pointer "backward", i.e., to the left, and "control F" moves it forward. "control P" goes to the previous line, and "control N" goes to the next line. Now each time one of these characters is struck, not only does the editor move its pointer within its buffer, but moves the cursor on the screen to the corresponding position in the displayed text. Thus, the user moves the cursor around via these controls, to the point where editing is desired, and inserts or deletes characters once he or she has gotten there.

A buffer (representing a segment being edited) can surely be longer than the number of characters which can be displayed on one screen. If an attempt is made to move the pointer out of the part of the buffer which is on the current screen, the editor will provide for moving the viewed "window" (the 20 or so lines being displayed) to a different set of lines, so that the pointer is always visible on the screen as the cursor. With intelligent display terminals, this can often be done by actually instructing the terminal's microprocessor to move existing lines around on the screen, up or down to make room for new lines which Multics will transmit.

Now this is only the basic idea. There are much better and more sophisticated commands available, such as "Capitalize the word at which the cursor/pointer points" or "Move the cursor to the matching close-parenthesis of the open-parenthesis where it is now", or "Delete the next 6 words" or "Put over here the 6 words I just deleted from somewhere else". Some notion of the flavor of these commands may be gleaned from the protoype command repertoire in the appendix. In each one of these cases, a visual indication of what was done is displayed immediately on the screen, not by leaving a record of the editing commands, but by modifying the displayed text on the screen. It is more akin to the notion of editing a text with a perfect pencil and eraser than with a computer. The user is always viewing the text as it appears after the last character he or she has typed. Thus, there is no need for the editor "print" command, which means "Let's see what I've done". You are always looking at what you have done. One moves the cursor to THE character or word one wants to change or delete, pointing at it directly, instead of trying to invent a context which specifies it uniquely, as in printing-terminal oriented editors.

The editor manages the screen with knowledge of its capabilities and limitations. Text neither rolls off the top as editing is performed, nor need be printed in hardcopy to view what one is working on. The terminal, the user, and the editor cooperate and interact in a fashion which is completely new to Multics.

---

(1) "Control B" is generated on a ASCII keyboard by pressing the "B" key while the "CTRL" key is depressed.

IV. Why do you need all those silly control characters?
They're not multicious at all.

It seems to make more sense that "a" should mean "enter an "a"" than "append". Since all the "good" characters (i.e., the printing characters other than #, \, and @) are used up in this way, we need control characters to say anything else. The default character assignments are reasonably mnemonic, e.g., "control B" for backward a character, "control E" for end of line, etc. Since this gives exactly 26 possible commands, and many, many more are desired, there must be multi-character escapes to express many of the commands. Thus, the ASCII "escape" (or "alt mode") key is used as a prefix to extend the number of commands, as is the "control X" character (1) in the default command assignment. Thus, the most common and useful commands are assigned to single-keystroke sequences (2) while less common ones require the "escape" prefix, and so on, until the least common require their full command-name to be typed (in response to escape-X, the default character for "read the name of an extended command and execute it").

The alternative to using "control characters" is to have "input" and "edit" modes, where the meaning of all printing characters changes in "edit" mode, as in qedx, edm, and TECO. This is one of the worst failings of these editors, as users happily delete lines by typing "del" in edit mode, write out segments named "ho", and insert sequences of "w", "q", and "pl1 foo" in their segments. However, printing-terminal editors are constrainted to printing characters, for the user must look carefully at his or her "substitute command" before unleashing it by typing carriage-return, in order to ensure that it will hopefully do the "right thing". This, of course is completely unnecessary if there is a blinking dot on the screen pointing at what you are about to delete or change.

All ASCII video terminals can generate control characters. There is no need to be able to see characters which are editor commands unless you don't believe that they are going to do the right thing if you "activate" them. All "delete" or "kill" type commands can be undone. If some command character typed deleted the wrong thing, one need only type "control Y" (for "Yank") to get it back. Since the effect of the deletion is

---

(1) All of the character-to-command assignments can be changed dynamically by any user at will, if he or she so desires. This goes as far as saying "when I type "~" I mean "E", because the "E" key on my terminal is broken."

(2) Not counting the "CTRL" key, on which one tends to hold one's little finger during other sequences than text-insertion.

viewed instantaneously, no further work is lost. Thus, there is no harm in simply commanding the editor to do what you want in real-time, undoing it if it was wrong. You don't ever want to see what you typed (with respect to editor commands): you only care what you have in the buffer, which you always see. Thus, you do not have to "plan" and "plot" editor commands, as with the conventional editors, before activating them.

Holding a "Control" key to achieve a different function of a key is no better or worse than holding "shift" to capitalize a letter. So much for control characters.

V. Can't intelligent terminals do this locally?

Well, some of them certainly can do some of it. Many terminal manufacturers provide a facility whereby you can edit the text on the screen via the use of specially marked keys on the terminal. Response is instantaneous, since only one user is using the terminal. When the screen is all correct, one can hit "transmit" and the screen will be transmitted to Multics.

This approach has several problems. The most serious is that there is no cooperation between Multics and the terminal. All changes made are made locally. If text is deleted, it is gone, and cannot be gotten back. You cannot edit any part of your segment except that which appears on the screen. No known terminal on the market provides for searching, word processing, or, in general, any form of text manipulation more sophisticated than the insertion or deletion of single lines or characters. The power and generality of the Multics software cannot be used.

Furthermore, since Multics does not cooperate in this venture, a random message or unexpected transmission messes up the displayed screen beyond repair; all work is lost, and Multics cannot regenerate it because it was not in on the editing. It is a situation as dynamic as setting lead type offline. Any unexpected situation spills the type on the floor.

Another good problem with the local editing approach is that not all terminals have the same local editing capabilities. Many very good and popular ones (such as the DEC VT52) cannot insert characters in the middle of a line, or move lines around locally. Using local editing, these operations are simply ruled out. Using real-time editing on the central system, it is a simple matter for the editor to simulate these operations in terms of the terminal's repertiore. At 1200 baud and better, the difference is barely noticeable.

Transmitting entire screens, or even lines, at line speed, places severe constraints upon the central system receiving end. Large buffers and split-millisecond allocations are required. This technology also requires high line bandwidth in the "transmit" direction at any speed. At lower speeds, the

time required to transmit a "screen" is simply annoying.

Local editing is not extensible. One cannot write editor "macros" for a read-only memory.

VI. Why don't you use those buttons with the arrows on them, and the "program function keys"?

Experience with this kind of thing has shown that one can edit faster if one keeps ones fingers on the normal typing keys of the keyboard, without moving them around for editing operations. It is partially this simple matter of "hand efficiency".

Not all terminals have the same set of special-function and program-function keys. The number, nature, and meaning of these keys is a function of the manufacturer and model of the terminal. By using normal keys and their "control" codes, the description and availability of editor functions is a specifiable in one place, the single document for the editor, regardless of what type of terminal is being used, what keys it has, or what functions it has available. Usage of the editor from a Honeywell VIP 7200, a Perkin-Elmer FOX, or a MIT-AI Knight TV Display is identical.

My sympathies do not lie with them that cannot understand that "control B" means backwards, for its lack of a left-pointing arrow, given that they understand the notion of "\f". The full capabilities of the intelligent display can be utilized without the use of random and variegated local keys; they are simply not useful in this context.

VII. What have you done?

I have designed and implemented a real-time display editor on Multics. Running on the MIT and CISL machines, it has, in the month since its inception, acquired an actual user community of about a half dozen. Display terminals are prevalent in the MIT Computer Systems Research and Programming Development Office groups.

The command repertoire and interface philosophy of this editor were borrowed from the "EMACS" editor running on the MIT AI/Mathlab PDP-10's. "EMACS" stands for "Editor Macros", for a screen editor implemented by Richard Stallman of the MIT AI lab as a set of "macros" (1) in ITS (2) TECO. Mr. Stallman is fully

---

(1) "Macro" is the term used in TECO and many other editors to describe user-coded programs and extensions. It is a poor term.

(2) For "Incompatible Time Sharing," the MIT-developed operating system used on the Artificial Intelligence Lab's PDP-10's.

aware of this endeavor, and most enthusiastic about it. Having used EMACS during the preparation of the 1978 offering of my MIT Lisp Course, I became acquainted with this particular interface, and enamored of it and what it represents. Nevertheless, EMACS is not unlike, in basic ideas, any of a half-dozen other state-of-the-art screen editors that come to mind, and neither MIT nor DEC has any proprietary rights to the particular commands and character assignments. The idea of real-time video editing is also in the public domain. The fact that I have preserved the interface of ITS EMACS has allowed an existant user community to utilize the Multics version of EMACS immediately.

EMACS is considered to be state-of-the-art. Cognoscenti have observed that no other video editor offers any particular advantages of substance. It is high time that Multics editing technology left CTSS-oriented editing and became state-of-the-art.

Multics EMACS, as my editor currently calls itself, is implemented in Lisp. I will discuss this below. It is extensible, programmable, self-documenting, and its code is transparently legible. It is already a powerful tool which I have used in my "normal" activities with great success. It already supports seven different makes/models of video terminal. It takes a skilled programmer about 10 minutes to construct a support extension for a new type of video terminal. The need to construct these packages in the editor as opposed to in Multics stems from the fact that Multics does not support video terminals with any kind of mechanism like the TTT with which printing-terminal extensibility is implemented.

The response to this project has been overwhelmingly enthusiastic from the user community. People have offered the opinion that it is high time that Multics had something like this. At sites such as USL, (which runs DD4000's exclusively, Multics EMACS' "favorite" terminal), the anticipated response can hardly be envisioned.

VIII. I have heard that you patch the FNP with hphcs_.

Indeed, this is true.

For asynchronous terminals, Multics is currently a totally line-oriented system. Character at a time interaction is essentially impossible. The FNP (1) will not transmit a line to Multics until a new-line (2) character is typed. The FNP-6180 interface protocol is organized upon the transmission of completed lines. It is precisely this technology which fosters the growth of line-oriented editors and premeditated substitute commands.

The Multics ARPANET implementation will transmit each character as it arrives, and wake up a the server Multics process. Since other hosts support character-at-a-time I/O, specifically the TIPs (3) the MIT community has had to resort to logging in to Multics from the MIT TIP or the PDP-10s in order to use Multics EMACS, because Multics itself cannot perform character-at-a-time input.

Thus, I have found it necessary, to create in the FNP (with the help of Larry Johnson, one of the local FNP experts) a CCT (4) which breaks on every character, and patch it into use for my channel every time I use Multics EMACS. I do this on the CISL development system during the time that "mini-service" is run. This causes every character to be treated like a "newline", and transmitted to the central system from the FNP. The response time, through the FNP, the central system interrupt side, the ring 4 program and back out again, manifests itself as a character-echo faster than LSLA echoing, comparable to HSLA echoing. Admittedly, there is little or no user load on CISL, and it takes quite a bite out of the system. This does not make it less useful or less good.

This "break and ship buffer on every character" mode is being proposed as an interim MCS extension to allow debugging and

---

(1) Front-end Network Processor, being a Datanet 355 or 18X.

(2) Carriage-return, line-feed, or either, depending upon various modes.

(3) A TIP (Terminal Interface Processor) is a node on the ARPANET which can be dialed up, like the FNP, from phone lines, and will allow the user to log into any host on the ARPANET "from" it. In the jargon of the ARPANET, the TIP is thus the "user" end of a "TELNET" connection, and the "foreign host" (i.e., the desired system) is the "server" end.

(4) Character conversion table, the hardware table in the FNP which tells an HSLA subchannel which characters to interrupt on

experimentation. It is planned as a tty mode for the near future. For demo and experimentation purposes, it will be completely adequate. It may even be totally adequate, and better DIA protocols may not need to be devised for some time.

The current line-oriented DIA (1) protocol involves three handshakes between the central system and the FNP to ship that line over. Negotiations to allocate a central system buffer, and negotiations to allocate a mailbox over which to negotiate the central system buffer are involved. This overhead is unacceptable, both in terms of number of interrupts, CPU time, and response time under load for highly interactive time sharing applications such as real-time editing. The DIA mailbox protocols were partly designed for NPS compatibility; these goals must be reexamined.

Other systems ship characters with process "destinations" at regular intervals (say 1/30 sec.) between front-end processor and central system in multiplexed buffers. Such a scheme for DIA transfer maintains the same throughput with complete character-at-a-time response.

There are other schemes to reduce the overhead of a DIA transaction, including the sending of short messages in the mailboxes as opposed to negotiated buffers. Other schemes involve the regular shipment to the FNP of the addresses of available buffers. The scheme proposed above is not as radical as it may first appear; it is not unlike the ring-zero demultiplexing currently being proposed for IBM 3270 support.

---

(1) Direct Interface Adapter, the communications path between the FNP and the central system.

IX. That's going to take some bite out of the system, isn't it?

Yes. You get what you pay for. If Multics is not responsive or fast enough to support this highly interactive application, we cannot truly claim we have a truly interactive time-sharing system. Current performance via the ARPA net on MIT lags three or four characters behind typed input. This, however, is going through the substantial overhead of the Network Control Program, the IMP dim, the packet-switched ARPA network, etc. Yet, these results are on MIT with "real users" logged in. It is felt that Multics response may not be as bad as some fear; if the TTY dim must be subjected to the same type of scrutiny and recoding that has been performed upon page control and the Disk Dim, then this is as good an excuse as any. People want functionality as well as performance for their computer dollars.

X. Why did you have to write it in Lisp?

The language Lisp is extremely attractive as a tool for the development of user-extensible, modular, efficient programs. It overcomes many of the disadvantages of block-structured languages such as PL/I, Algol, and Pascal.

In Lisp, I can make functions 3 or 4 lines long that I can call from any place in the entire subsystem with a call overhead of about six instructions, and a stack frame overhead of usually 0 to 4 words (on the Lisp stack, of course). In PL/I, for example, I am forced to make the choice between external and internal procedures. The call overhead for external procedures is unacceptable, being near two dozen instructions including the callout, entry, and return operators. The minimal 64-word stack frame is also unacceptable. For trivial functions, such as "go forward a character", the external procedure must be ruled out.

The internal procedure is similarly problem-laden. By definition, it may only be used in the procedure in which it is defined. This means that either its text must be duplicated, lexically and in object code, in all source modules that wish to use it, or one very large source module must be used. In either case, the current implementation causes the stack overhead of the internal procedure to be paid for even when it is not being called, adding to the frame of the procedure in which it lives. In the first case, modularity is sacrificed because everything must be recompiled or changed if one internal procedure changes. In the second case, maintainability is sacrificed, because one gigantic procedure must be recompiled for the most minimal change. Furthermore, internal procedures cannot be traced.

In neither case can user extensions call these internal procedures easily, or with any kind of transparency or

modularity.

The programmer efficiency of most block-structured languages also leaves a great deal to be desired. Most characters in a source program are syntactic constructs of the language, not user variables and functions. The weight of mandantory declaration bears heavy upon the smallest source module of most subsystems. Lisp object code suffers not one instruction for the lack of these declarations. Optional non-generic operators address the data-type issues adequately.

The Lisp environment provides a "process" not unlike the Multics process environment, where any piece of the subsystem may be called from any other. User code can call any function in the subsystem. (1) Unlike PL/I, this does not reduce the efficiency of the subsystem. The Lisp environment is finely tuned for allocation in well-defined increments, as this is one of the tenets of the language. As dynamic an application as a real-time editor relies heavily upon this. This path is more efficient than the standard area path.

A large Lisp subsystem can be debugged incrementally, as functions are added (either by the developer or a user extending it). Not only is recompilation/rebinding (worsened by the procedure call issues outlined above) not necessary, but the support of an interpreter is available for debugging. No other compilable language on Multics has such a facility.

Where function call is still too expensive, Lisp provides a macro facility, whose invocations look identical to function invocations. It is tremendously powerful, and useful. No other compilable language on Multics provides any macro facility.

Many of the traditional complaints about Lisp are founded in darkness. For example, be aware that production subsystems in Multics Lisp are not interpreted, but compiled, by one of the finest Lisp compilers available. The Lisp compiler (2) produces standard Multics object segments to run in the Lisp environment, offering a performance improvment of over 100 over interpreted code. Compiled and interpreted functions can call each other freely.

"Lots of Irritating Single Parentheses", as the acronym

---

(1) Whether some limitation on this is desirable is another issue.

(2) A particularly interesting program about which I have written an extensive document of probable interest to those with an interest in compiler theory. It is available upon request.

"Lisp" (1) has been accused of denoting, are a problem only when proper editing tools are not available. The parenthesis and S-expression balancers of EMACS (and Multics EMACS) reduce the grief of Lisp editing by a factor of 10. Once the hang of it has been acquired, it is found that the time to prepare a Lisp function (say in the editor) that works can be measured in seconds, while a comparable PL/I preparation might take 20 minutes, including all the declaring, compiling, binding, debugging etc.

The efficiency of character-handling is also not an issue. Multics EMACS contains a small amount of assembler (2) code for inserting and deleting characters from the active line. Note that PL/I cannot utilize an mrl instruction to open up a line, either. Calling a LAP program from Lisp still takes the same 6 instructions: PL/I still needs a dozen or more.

Lisp is not the optimal language for all tasks. Dealing with machine objects already laid-out in storage is not particularly efficient. The representation of data objects does not lend itself well to packing bits and characters, etc. The structure concept of COBOL and PL/I is superior for these applications. But in the editor, neither of these are issues.

XI. Does this mean that we have to support Lisp?

Yes and no. Support means a lot of things. We do not need to promote it as a product, or as an up-front user language at this time. These things can come in time.

We must install the Lisp environment support in whatever library this editor would be installed in. We must be responsible for fixing bugs if necessary, and making minor extensions as needed. These commitments do not take much personpower.

We do not need to distribute an "Orange Cover" Honeywell user document, although that option remains open to us. Documentation of Multics Lisp is available to us, in on-line form, and the creators of this documentation have offered it to us. This documentation is currently available internally, and can continue to be so.

We do not need to create marketing documentation at this time. We should train some more developers within the Multics software group at this time: a tutorial document prepared by me is available internally. It should not be necessary at

---

(1) Actually for "List Processing Language".

(2) Actually LAP, the Lisp Assembly Program, a type of ALM which produces Lisp-loadable object segments.

this time to train supporters outside the development group; this issue can be addressed later.

Some documentation must be distributed in order to enable potential writers of editor extensions to code them effectively in Lisp. This is not a large problem.

We feel that the time spent supporting Lisp in whatever capacity will buy itself back in an increased number of development tools. There are already some private tools in Lisp of some importance. Any support of Lisp would vastly augment their status.

The Multics Lisp implementation is an implementation of MACLISP, one of the more well-thought-out and complete implementations of Lisp available. It is completely operative, and has no known major bugs. Major extensions are not needed at this time. We can have it now.

XII. What about "editor macros"?

It has been observed by those who have addressed this problem before that editors which try to offer a single language for interactive editing and macro-writing develop a compromise which is inadequate at both. The qedx interface is not unreasonable for a line-oriented printing-terminal editor. The commands are natural, and simple, for that application. However, developing "macros" (complex pre-written functions) out of these functions is a feat of such magnitude that a former Multician pinned to his door a qedx macro so complex that it could actually play tic-tac-toe. There are no facilities for program control, variables, subroutines, etc., in this language. It is not a good programming language. Various attempts to add these features are ill-conceived, and do not make it any more so suited. The qedx language is still a human-to-editor imperative language by which to negotiate changes to a segment.

The other end of the spectrum is TECO, either on Multics or the PDP-10's where it was born. In an attempt to provide functions suitable for program-writing, such as loop constructs, character-at-a-time movement, etc., TECO requires someone editing a segment to devise little programs to do something as simple as "change all foo's to bar's". Teco is oriented towards program writing; the program-instructions are really too primitive to be used as commands in a line-oriented environment. On the other hand, the single-character names of the commands, a feature to allow use as an editing language, cause TECO programs to be notoriously abstruse. Thus, TECO fails at both.

It is the opinion of those who have previously addressed this issue that there should be two languages

associated with an editor; the keystroke language by which a user edits his or her segment, and the language for writing macros. This does not constrain the programming language to be terse, nor the keystroke language to be primitive. Thus, ITS EMACS was implemented via TECO, as a programming language, but with an newly designed editing language optimized entirely towards editing.

The Multics EMACS editing language is the provided list of control characters and escapes.

The Multics EMACS programming language is Lisp. Lisp is a complete and powerful programming language, with all of the features one would expect in a higher-level language. The Multics EMACS programming language is enriched by the functions provided in the editor environment. Many of these functions are the ones normally invoked in response to keystrokes, such as the function "backward-char" which is normally invoked when "control B" is typed. If you know what "control B" does, you know what invoking "backward-char" does. I contend in dead earnest, by experience, that a "macro" written in Lisp for Multics EMACS is more readily comprehended by one who does not know lisp than a comparable macro in TECO or qedx would be by one versed in these languages. So transparent is this Lisp code. This is largely due to the lack of syntactic overhead in Lisp, and the aforementioned Lisp macro facility.

I have devised and debugged Multics EMACS editor extensions (a better term than "macros") in real-time, by typing them into a buffer, staring at it, saying "OK, editor, now load it" and now "try it". If it doesn't work, your are still looking at it. Edit it, try it again. You never leave the editor. You see the function and its results in front of you. It is quite effective.

Via this technology I have devised esoteric functionality such as hitting a control-character which fills in the declaration for the PL/I subroutine name I just typed, and another which allows me to "edit" the listing of a directory, deleting the segment whose name I have pointed the cursor at, and editing the display in parallel.

Any user-function, or any editor-provided function, can be hooked up to any key, or invoked explicitly by its name.

I fail to see why any person who can write an editor macro in qedx or TECO would have problems, given a well-documented starting point, writing editor extensions in Lisp. Knowledge of the internal organization of the editor is not necessary. What is more, the display is managed automatically by the editor. User extensions (or builtin functions, for that matter) are not aware of the existence of the display; whatever transformations upon the buffer were performed by the user code

or the editor code, these changes will be made automatically to the screen when they are all done, via the technology known as "redisplay". All in a terminal-independent fashion.

XIII. What about "The Editor"? Is this instead of the proposed qedx extensions?

If you have a display terminal, yes. I cannot envision any reason why anyone would want to use qedx or any of its derivatives given the capability for real-time video editing.

If you don't, probably not. Although EMACS does work on printing terminals, I would prefer qedx, and given that, the current research into conventional editing is well worth it. The only advantage to using EMACS on a printing terminal is the large number of user extensions that people will have developed. ITS EMACS moves the printing-head around instead of the cursor, and it is curious to watch.

XIV. Where can I see this thing?

In Cambridge, it is best demonstrated on the CISL Delta-Data 4000 during CISL service, 1:00-3:15 PM. If you have any kind of display terminal and ARPANET access, you can probably use it on the MIT machine. Contact me. (HVN 261-9330, or 617-492-9330, or Greenberg.Multics on MIT, CISL, or System M, or Greenberg @ MIT-MULTICS (ARPANET), or BSG @ MIT-AI or MIT-MC, or care of Honeywell Information Systems, 575 Technology Square, Cambridge, Mass., 02139.)

I hope to have it operative on the VIP 7200 in Phoenix, System M, in the very near future.

Current repertoire

The following is a command list that I hand out. It is nowhere near complete, in terms of what I plan. It is here simply to give an idea of what some types of possible things are. This is the current repertoire of the implmentation.

The Multics real-time editor is an interactive, display-oriented text-preparation and editing facility designed after the EMACS TECO macros of R. Stallman and others on the MIT-AI PDP-10.

This is an elementary list of editor commands. Almost all printing characters represent themselves and go in as text; Carriage-return or linefeed may be used to terminate lines. The symbol "^" means "control" - "^A" means depress the "a" key while the "CTRL" key of the terminal is depressed. The symbol ESC means that the key labelled "ESCAPE" or "ALTMODE" should be depressed and released prior to pressing the next key. I.e., ESC-D means the two-character sequence, ESCAPE D. (Lower case can be used, i.e., ESC-d).

The cursor is considered to be the LEFT edge of the blinking cursor, i.e., when in the leftmost column, it is considered to be before the first character.

^A Go (move cursor) to beginning of line.
^B Go Back one character.
^C No-op. See ^L.
^D Delete character at (to right of) cursor.
^E Go to end of line.
^F Go Forward one character.
^H This is a backspace- do NOT USE.
^I Same as a TAB.
^J Same as a linefeed.
^K Kill to end of line, except when already at end of line, delete the linefeed (merge lines).
^L With numeric arg (see below), kills that many lines. Redisplay the screen. Use if terminal or editor starts losing, or somebody sends you a message, etc. On FNP Multics, follow this by ^C, because FNP will not transmit ^L alone.
^M Same as carriage return.
^N Go to next line, same horizontal place.
^O Open up space, insert a linefeed, move cursor back. See ^U. ^U^U^O for instance will open up 16 lines.
^P Go to previous line, same place.
^Q Quote the next character, i.e. insert it literally, as ^Q# to get a pound sign in. Same as \.
^R Reverse search. Leave cursor positioned before matching string, don't move cursor if not found. See ^S.
^S Search. Will prompt at bottom of screen for search string. End the search string with ESC.
^T Twiddle (transpose, interchange) the last two characters typed. like, I like Mutl^Tics becua^Tse... etc.
^U Multiplier. When not followed by a number, multiplies the next command by 4 for each use. I.e., ^U^D deletes 4 chars. ^U^U^D

deletes 16. With a number, uses that, i.e., ^U13x inserts 13 x

^V not implemented yet.

^W Wipe (kill) all text between cursor and the-mark. Can be retreived with ^Y.

^X Control X commands are two-character sequences, listed below.

^Y Yank (retrieve) killed text to cursor. Unkills last killed word, line, or region (^W). With an argument, goes that many killings down a 10-position ring-buffer of old killings.

^Z does a Multics Quit.

^@ Sets the-mark to be where the cursor is now.

\ Causes the next character to be inserted literally.

@ Kills all the text on the current line.

Deletes the previous character (before the cursor, which is usually the last character typed, like in normal Multics.)

ESC-[Go to beginning of paragraph, right before first word.

ESC-] Go to end of paragraph, end of last line. Runoff control lines count as paragraphs.

ESC-< Go to beginning of buffer.

ESC-> Go to end of buffer.

ESC-B Go backward one word, leave cursor before first character of it.

ESC-C Capitalize the current or last word, move to after it.

ESC-D Delete the word to the right of cursor, and all whitespace between cursor and it.

ESC-F Go forward one word, leave cursor after last character of it.

ESC-G Do to point/mark region what ESC-Q does to a paragraph.

ESC-H Set point and mark around the current paragraph.

ESC-L Lowercase the current or last word, move to after it.

ESC-Q "Fill" the current paragraph, like runoff with ".na". With argument (i.e., ESC-1 ESC-Q), fill and adjust like runoff with .fi and .ad. See ^XF to set fill-column.

ESC-U Uppercase the current or last word, move to after it.

ESC-W Like ^w, but doesn't wipe, just puts in kill ring.

ESC-X Prompt for the name (and args) of an extended command. See below.

ESC-Y I don't like what I just ^Yanked. Get rid of it and yank the previous thing in its place.

ESC-\ Delete all whitespace surrounding cursor on current line.

ESC-# Delete the word to the left of the cursor, and all whitespace between it and cursor.

ESC-^A Go to beginning of Lisp function, i.e., last line with open paren in first column.

ESC-^F Skip over exactly one balanced S-expression, including parenthesized lists. This is VERY powerful, and may be used to balance parentheses in PL/I or anything else. Diagnoses missing ")"'s.

ESC-^N Skip to end of current Lisp list. Used internally by ESC-^F.

ESC-^O Break line at this point, indent new line like this line.

ESC-ESC Prompt for string for Lisp to evaluate. For hackers only.

ESC-(number)

Causes the next command, if one of the following, to be repeated that many times. I.e., ESC-35-^D deletes 35 characters. ESC-4-m inserts 4 m's. Same as ^U when used in this way.

Good for ^B ^D ^F # ^N ^P ^O ESC-B ESC-F ESC-D ESC-#

Also tells ^Y how many back to yank.
Also tells ^K how many lines to kill.
Also tells ESC-Q whether to adjust or not.

^X commands:

^X. Set "fill prefix" to what's between beginning of line and cursor. The "fill prefix" is inserted automatically by linefeed, ESC-Q/ESC-G, and autofill. It is also prepended to lines that are yanked, if they follow yanked linefeeds.

^X^R Read a file into buffer, leave you at first position of first line. Prompts for file name. Terminate file name with CR or LF.

^X^W Write out buffer to file. Prompts for file name. Terminate with CR or LF.

^XB Switch to new buffer, or old one. Prompts for buffer name, which is terminated by CR or LF. Buffer name shows at bottom of screen.

^XF Set "fill" column for ESC-Q and speedtype/autofill stuff to horizontal position where cursor is now.

^X^B Shows listing of buffers and pathnames. ^XB somewhere else to get back to what you were doing.

^X^L Lowercase all letters between cursor and the-mark.

^X^S Write out buffer to last file read or written in this buffer.

^X^X Exchange the cursor and the-mark, to verify what you are getting into before typing ^W.

^X^U Uppercase all letters between cursor and the-mark.

^X^M Prompt for a Multics command line. Terminate with CR or LF. Multics commands that produce output may well screw up your display, may have to ^L.

You may edit in the minibuffer (the bottom of screen prompting area), ^Y'ing things from elsewhere, etc. If you try multiple lines, you only see 1 line at a time. Use ^QCR or ^QLF to get linefeeds in when linefeed is the terminating character.

Extended commands - invoke by ESC-X. Type command and args, if any, in minibuffer, where it puts you. Terminate by CR or LF.

replace Global substitute. Will prompt for two strings, terminated by ESC. Replaces all occurences of first string by second, leaves you after last occurence.

speedtype Enter speedtype (word abbreviation) mode.

setab Set a speedtype word abbrev. E.g.,

```
setab bsg Greenberg
```

Can take multiple pairs of args, if that's convenient.

fillon Set auto-fill mode. Speedtype mode sets this automatically.
filloff Turn off speedtype and autofill modes.
put Takes one arg, a "variable" name. Same as ^W, but puts text in that "variable" instead of on the kill stack. Use getback to get it back.
getback Takes one arg. Like ^Y, but yanks back the named "variable" which is its arg.
lvars List names and lengths of all variables ever "put".
quit Exit the editor.

The editor accepts META characters from AI TV's, and does TELNET break/interrupt processing.

This document, of course, was prepared with what it describes.

CBI/1/17
GG

MTB- 418

Date: 7/11/79

To: Distribution

From: Bernard S. Greenberg

Subject: The Echo Negotiation Papers

This MTB consists of the two design documents for the "Echo Negotiation" schemes, Ring Zero and FNP. Echo negotiation is a general scheme for allowing synchronization of output with echoing of input in a multilevel communications processing hierarchy. Such synchronization is necessary in a managed-video environment, for echoing of input is conditional upon processing of previous input, and must be performed at places on the screen determined by output from previous input.

The need for these schemes in Multics is motivated by the need for performance amelioration of the experimental Emacs editor, which maintains such a managed-video environment, and has need for such synchronization. In its first implementation (March, 1978), Multics Emacs achieved this synchronization by receiving all characters in ring 4 as they were typed. This was inordinately expensive, requiring a wakeup and process loading, as well as the running of a user process and multiple calls through the TTY Dim, for each character typed.

In January, 1979, the scheme described by the first of the two papers contained herein was implemented, shifting the principal responsibility for echoing echoable typed input from the ring 4 code (Emacs) to the ring zero interrupt side of the Multics TTY software. This reduced the number of wakeups and loadings for the inputting of text from one per character (50 or so per line) to two per line. This implementation was shipped as part of Multics Release 7.0a.

As some experience with this scheme was gained, it was predicted (given 6 milliseconds, worst case, for MCS interrupts to the mainframe) that 100 Emacs users would entirely saturate one CPU with interrupts (assuming text type-in at 5 characters a second). This appeared to be the most immediate and most crushing bottleneck in Emacs performance. Thus, a scheme was devised to eliminate per-character mainframe transactions where possible, by having the Front End Processor (FNP, FEP, 355, 66/32, 66/78, etc.) echo characters for typed input. Correct implementation of this facility requires synchronization protocols to carefully control the circumstances under which echoing is performed: the circumstances under which it starts, and stops. A now-operative (July, 1979) implementation of this idea is described in the second paper.

Multics Project internal working documentation. Not to be distributed or reproduced outside the Multics project.

As implemented now, a four level hierarchy of echoing software exists:

I	Emacs Lisp Code
II	Emacs MCS interface (e_pl1_)
III	Ring Zero MCS (TTY Dim)
IV	FNP (potentially, other "multiplexers")

The top level (the Emacs editor) asks for and receives non-echoed characters from the lower (higher-numbered) levels, until it enters the "negotiated echo state", by being at "rest" at the end of a line (and certain other conditions met: see the first paper). At this point, Level I calls level II, passing as a parameter the length of the line left on the screen to the right of the cursor. Ultimately, Level II returns to Level I a collection of N characters, the first E (E possibly zero) of which were echoed.

Level I does not care which level echoed the E characters, all it cares is that they were (or perhaps were not echoed). Level I will process all the returned characters that were not indicated as echoed, processing them as Emacs "commands", including echoing those that should be echoed but were not indicated as already echoed.

The interface between Level II and Level III, and between Level II and Level IV, is virtually identical. Each returns what characters it has buffered if it has characters buffered, telling how many were known to have been echoed, and in this case calling no lower level. If a level has no characters, the next lower level is called, passing the screen-length left. That lower level returns characters, some of which it indicates as having been echoed, and the rest not. The upper level considers the lower level's echoage as part of its own, and processes the remainder to attempt to echo the first E' characters after the returned echoed string. In this way, any lower level can "punt" echoing for arbitrary or internal reasons, being assured that echoable characters will be echoed by some level, and optimally, one as far down (and thus, efficient) as possible.

The "multilevel echo hierarchy" also provides for lower levels that do not support this echo protocol: the upper level will simply echo, as the lower level has returned no echoed characters. Similarly, lower levels yet (i.e., programmably-echoing terminals or concentrators) can be added in an integrated and consistent fashion.

Levels also ship "break tables" down the line to each other. It can be shown that these tables can only be requested to be shipped when the entire system is in the non-echoing state.

The current FNP echoing is not more consumptive of FNP time than echoplex: no number of asynchronous lines running echoplex has ever caused a shortage of FNP time; thus, there

appears to be little motivation (for the FNP, at least) of devising schemes to have "echo-negotiating terminals". Attempting to do so even introduces new problems (see the second paper). For common-carrier net (e.g., TYMNET) access to FNP channels, this is still an interesting problem.

(END)

Interrupt-time Echo Negotiation

BSG 1/21/79
revised 1/27/79
ESC-Q'ed to ^XF = 72 6/30/79

Interrupt-time echo negotiation is a proposal to modify the interrupt side of ring-0 MCS to conditionally "echo", or send back to the user terminal, characters input from the keyboard. The goal is performance improvement of Multics Emacs (or any other conceivable real-time editor implementation on Multics), and increasing the apparent responsiveness thereof many fold. Complete transparency to the Emacs user as well as the Emacs extension-writer is a design constraint.

Programs which attempt to manage a display terminal intelligently cannot allow the terminal itself to unconditionally print characters as they are typed. As long as type-ahead is allowed, characters must be placed on the screen at positions determined by the central system's interpretation of previous characters. Until all previous characters have been processed by the central system, and the terminal's cursor repositioned appropriately, characters may not be typed out as they are typed in ("echoed"), whether it be by the central system, the FNP, or the terminal itself. Thus, the use of "local copy" (unconditional terminal echoing) is ruled out in a display-managed environment.

The current Emacs implementation receives each character from the terminal via calls to the Ring 0 tty dim; several may be read at once. If Emacs was blocked waiting for input, only one wakeup will be sent. If Emacs was not blocked, no wakeups will be sent (this is an old feature of the tty dim). If Emacs can process characters as fast as they are typed, one wakeup, one process loading, one read (get_chars) call on ring zero, and one write (put_chars) call on ring zero will be made for each character. This shows up as a tremendous CPU and paging consumption in the user ring (and user-initiated calls). If Emacs falls behind, fewer calls are made and more characters are read and written per each call to ring zero, and no wakeups occur until Emacs actually catches up. Strangely enough, it is thus much more efficient when it falls behind.

The case of "just typing in text" is, statistically, a very important one, one of the most frustrating when the system is slow, the greatest cause of expense (largest number of characters per second) and the one most amenable to optimization. If no typing errors are made, it is the case where Emacs is doing the least for the user for the greatest expense. But when typing errors are made, Emacs excels here. "Just typing in text" means adding "trivial" printing characters (i.e., "a", "$", "/", " ", but NOT backspace, rubout, control characters, carriage return, etc.) at the end of a line. Included in this case is the typing of "minibuffers" (responses to Emacs-generated prompts). This case is what is dealt with by "input mode" in QEDX or EDM. It is one of the things that Emacs does best, but one of the most expensive.

The proposal is to optimize the "just typing in text", known in Emacs jargon as "EOL (end of line) self-insert" by having the

interrupt side of the tty dim echo characters when Emacs deems it appropriate. The effect of this will be that, once Emacs is caught up, characters will be echoed (and thus appear on the screen) as soon as they are typed. Emacs will not be woken up (or loaded, or run) until some "break condition" occurs, as defined below. The response time for EOL self-insert echo will only be a function of interrupt response (milliseconds even on a heavily loaded system) instead of process-loading response (which can vary from milliseconds to seconds).

A break condition, in this context, means a situation where ring 0 can no longer echo characters. Such break conditions are:

1) Enough characters have been echoed to fill the screen (terminal) line. We cannot know where to put the next character; although it may be simply at the start of the next line (this is an avenue for more future optimization), a complex redisplay involving moving the entire screen around may need to be done before the \c and continuation line can be displayed.

2) A non-echoable character is received. The ring zero dim will be supplied with a bit-table by Emacs of echoable characters. All control characters are non-echoable. They are editor commands, not characters to be printed on the screen. Ditto #, @, etc. Similarly, tabs and backspaces require special redisplay action.

3) The interrupt side can no longer echo, due to inability to allocate a buffer bound for the FNP, or other internal problem.

4) The person at the terminal hits QUIT (which also causes a resetread).

When a break condition occurs, Emacs (if blocked) will be woken up, and echoing will cease. Characters will continue to accumulate in the normal breakall-mode fashion until Emacs calls in and reads them. A return parameter of the read call is the count of echoed characters; the total count of input characters has always been a return parameter.

Emacs will process all characters read, as usual, inserting "trivial" characters in its buffer or minibuffer as it has always done. However, the echoing of "trivial" characters processed as EOL self-inserts by ring zero will be suppressed.

This implementation is believed to be race-free. Some of the ideas in it are due to J. Spencer Love and Jerry Stern. Earl Killian and Dave Moon contributed a few improvements, too.

A new entry point to the ring zero tty dim will be defined, hcs_$echo_negotiate_get_chars (see the include file at the end of this null). It is like the normal get_chars entry of the ring 0 dim (hcs_$tty_read), plus extra. Before describing the differences, it is critical to note some features of both these entrypoints:

1) The entrypoints are called with a device index, and a buffer and its size, and read characters into the buffer. The count of returned characters is returned.

2) If no characters are returned, a flag is set in ring zero, such that a wakeup will be sent when characters "are available". If characters are returned, this flag will not be set, and thus no wakeup will be sent, even when characters become "available", unless another call is made in the interim which finds no characters available.

   For normal tty_dim get_line calls, "available" means a complete line has arrived in ring zero. For get_chars calls in breakall mode, and echo_negotiate_get_chars calls in breakall mode, this means any characters having arrived (from the FNP) in ring zero at all.

   There is no way the input side of the tty dim ever sends wakeups (other than QUIT) other than this mechanism. The user ring is allowed to go blocked (i.e., expect a wakeup), if and only if one of these calls returned no characters.

The new entrypoint will, if no characters are available, return this fact, set this "wakeup flag", and set a bit in the wired terminal control block (WTCB) placing the terminal in "ring zero echo state". This means that ring zero's interrupt side will echo characters until a break condition occurs. This call-side operation will be locked against the interrupt side via extant tty dim mechanisms. Thus, the determination that no characters are available is a unitary operation.

If characters are available, all available characters will be returned. A count of characters that have been echoed before the break condition was encountered will be returned: call that count <n>. The wakeup-flag will not be set. The dim will not go into ring zero echo state. Emacs will be expected to process the characters, generating any redisplay necessary. The first <n> of these characters have already been echoed by ring zero; Emacs is not to echo them. Emacs is expected to call back when these characters have been processed, and all redisplay output from the the interpretation of these characters queued to the FNP in ring zero (via having made sufficient calls to output all redisplay output). At this time the terminal is NOT in the ring zero echo state; characters are not being echoed by ring zero.

When Emacs calls back, it will either be via the normal get_chars or the echo_negotiate_get_chars entries. If characters are available when Emacs calls back, thus, <n> will be zero for the new call, and Emacs is expected to echo all characters as necessary. Only when Emacs "catches up" will the new call enable the ring zero echo state.

Emacs will call the new entrypoint only when it describes itself as being "at the end of a line", i.e., amenable to EOL

which must be a self-inserting printable character (viz., not TAB), will be supplied by ring four, along with a count of necessary pad characters for this operation. On some terminals, this might be the sequence for "backspace, space, backspace", but on more sophisticated terminals, "backspace, kill-to-end-of-line" is preferred, which is a terminal-dependent sequence. A flag is provided as to whether characters deleted by the interrupt side in this fashion should simply be removed from the input buffer transparently or placed in the buffer with the "rubout" character; this latter alternative allows the ring four program to display to the user the "last 50 characters typed", often a useful ability. Two rubout characters are allowed, normally supplied by ring four. Pound-sign (#) and ASCII rubout would be the usual choice, but this is completely dependent on user action setting key-bindings, etc.

Note that the ring zero interrupt side, if doing "rubout" processing, can never delete a TAB or any (other) character it itself did not echo. If the dim is not in the echoing state upon the receipt of a rubout, or all accumulated, echoed characters have been rubbed out, the dim must wake up Emacs and suspend echoing, if on.

Any call to the normal get_chars entry disables ring zero echo (until the next call the echo_negotiate_get_chars that enables it). Within Emacs, as shown above, it should never be the case that it should need to. Yet, unexpected Multics action (i.e., a fault) causing ultimate return to command level, should turn it off. The first get_chars or get_line call that is made will thus do this. There is also an esoteric case involving the Emacs interrupt system (!), which is used for console messages and mode-line time displays, etc., which requires explicit disabling of ring zero echo, as cursor position will be changed even though no characters other than EOL self-inserts were entered from the keyboard. This problem is fairly hirsute. For detail on it, see

```
>udd>m>bsg>r0e.inter.text (MIT-Multics)
AI:DLW2;BSGR0I >    (MIT-AI ITS)
```

The scheme above has been implemented and tested (other than the interrupt interaction), and works as expected. It has not yet been metered.

There is a negative effect of moving any echoing out of the body of Emacs: by virtue of the same performance improvements that reduce the mean page residency time, the likelihood of the needed pages of Emacs being in main memory when needed, i.e., echoing stops, is reduced. Although this reduces the load on the system, and the cost to the user, the response time degradation when the bulk of the editor has to be paged in became quite noticeable the last time a paging/CPU performance improvement was implemented for EOL echoing, and this effect will surely be significant here too. On a very lightly loaded system, or hopefully, one where many processes are using Emacs, this effect will hopefully be ameliorated.

self-insert echo. Otherwise, the regular get_chars (hcs_$tty_read) entry will be called. Thus, inserting text in the middle of a line (which may involve "hard" redisplay) will not benefit by this optimization. Similarly, when ring zero echoing has been stopped due to typing a "prefix character" (say, ESCAPE or \), the next call will be to the regular get_chars (the motivation here is that ESC-A is an editor command, the A should not be echoed). This simply falls out of the implementation as so far described. Thus, only when Emacs states "I am at top-level command-level at the end of a line" is ring zero echoing enabled.

If the user can type characters as fast or faster than Emacs can process them, all echoing will be done by Emacs in ring four. By hypothesis, Emacs will never go blocked, and thus never be woken up, or need to be loaded. It will be processing characters in batch mode, reading and outputting many at a time. If the system is so heavily loaded that this is the case, that we cannot even get the sheer cpu time to process these characters, we are in deep trouble, which is to say, no worse than today. Unless this is the case, by hypothesis, Emacs will eventually catch up. Once it is caught up, ring zero will echo characters until a break condition occurs, and then we are waiting on Emacs to be loaded, process the characters, and ultimately revert ring zero echoing. Compare this to today where typing ANY character will wake up and load Emacs to echo it; the gain should be clear. I am pleased and proud that even under these adverse circumstances today, Emacs responds elegantly and rapidly most of the time.

A new order to the ring zero tty dim will be provided to supply a table of which characters will cause a break condition. This table will be set in wired ring zero storage by explicit call any time Emacs changes character bindings such that the self-insert/non-self-insert status of characters changes. This call will be made right before the next call to echo_negotiate_get_chars, once all such binding changes have been determined. There is no race here; this table can only affect characters that have not yet been processed by the ring zero interrupt side. Characters already in ring zero, processed, will be returned. None of them could have been echoed (perhaps falsely) by ring zero, because the previous call to ring zero must have returned characters. This must be the case because Emacs did something other than a self-insert, namely, something which involved at the LEAST switching some character bindings. Thus, the previous call to ring zero could not have been a zero-character-returning echo_negotiate_get_chars call, because it returned the commands that caused Emacs to "take non-trivial action". Thus, ring zero cannot, at the time of the table-set call, be echoing. Characters not yet typed, or queued on the channel lock in ring zero, in the FNP, or in transit to the Multics interrupt side are equivalent; they will be echoed according to the new table.

The info structure for the new order includes information for future implementation of rubouts (delete the last character from the screen, and ultimately the Emacs buffer) by ring zero. Thus, the most common typing correction will be processed without waking up ring four. The preferred sequence for removing the last character from the screen,

It has been suggested that inserting characters into the middle a line could also be optimized, placing a terminal in "insert mode" while ring zero is echoing. This is only useful, however, for terminals that support this feature, and requires management of the terminal's firmware state. Although common, this operation is nowhere as statistically significant as EOL echo. No such facility is planned for the near future.

One is tempted to ask as to why this echoing is not best done in the FNP; in truth, it would be much more efficient to do it in the FNP. Yet, the critical timing races which are solved by interrupt side versus call side locking in the above implementation are not amenable to such solution were the FNP doing the echoing. In fact, it is extremely non-trivial to devise a set of protocols such that all this works in that case. Lee Parks has proposed a set of protocols similar to the window managment and packet numbering of the CHAOS net which seem to offer some hope; Jerry Stern has devised similar, and these are under investigation, but implementation does not look imminent. If this sort of thing can be made to work, the door is open to pushing echo further down the line, to customized terminals. However, I cannot imagine that given any reasonable amount of FNP power, it would be necessary to resort to this for optimization of EOL self-insert redisplay. Moving editing functions involves extreme levels of complexity which dwarf any of the issues raised above.

But that's another ball game.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
/* BEGIN INCLUDE FILE mcs_echo_neg.incl.pl1  Bernard Greenberg 1/20/79 */

/* This include file defines the callable entrypoints and argument data
   structures for ring 0 echo negotiation */

dcl  echo_neg_datap ptr;
dcl  echo_neg_data_version_1 fixed bin static options (constant) init (1);

dcl 1 echo_neg_data based (echng_datap) aligned, /* Echo negotiation data */
    2 version fixed bin,
    2 break (0: 127) bit (1) unaligned,          /* Break table, 1 = break */
    2 rubout_trigger_chars (2) unal, /* Characters that cause rubout action */
      3 char char (1) unaligned,
    2 rubout_sequence_length fixed bin (4) unsigned unaligned,
    2 rubout_pad_count fixed bin (4) unsigned unaligned,
    2 buffer_rubouts bit (1) unaligned,
    2 rubout_sequence char (12) unaligned;  /* Actual rubout sequence */

 dcl  hcs_$echo_negotiate_get_chars
      entry (fixed bin (35), ptr, fixed bin, fixed bin, fixed bin,
      fixed bin, fixed bin, fixed bin, fixed bin (35));
 /*
```

```
call  hcs_$echo_negotiate_get_chars
      (devx, datap, nelemt, offset, NRETURNED, NECHOED_RETURNED, screen_le. ,
           STATE, CODE);

   END INCLUDE FILE mcs_echo_neg.incl.pl1 */
```

(END)

FNP Echo Negotiation -- 6/30/79, 7/11/79 -BSG
(>udd>m>bsg>fnp-echnego.text, MIT-MULTICS/System M)
(AI:DLW2;BSGFNE >, AI ITS)

This paper describes the scheme devised by Jerry Stern and myself to move the responsibility for "trivial Emacs echo" into the FNP. Lee S. Parks (MIT at the time, Lawrence Livermore Labs now) and Earl A. Killian (Bolt, Beranek, and Newman) also contributed ideas related and leading up to the current proposal.

The goal of this scheme is to eliminate the need for a Multics interrupt for every character received when the only action of Multics would be to accumulate and echo (send back to the terminal) that character. (Such "conditional negotiated echoing" is necessary in any managed-video environment, it is not something Emacs-specific). This echoing is currently performed by the interrupt side of the ring zero teletype dim (MCS, Multics Communications System) when feasible. That scheme, now part of Multics (as of Release 7.0), is described in

>udd>m>bsg>r0-echnego.text
(MIT-Multics and System M, Phoenix)
DLW2;BSGR0E >
(MIT-AI PDP-10)

That document is prerequisite reading for comprehending this paper. The present scheme is built upon that scheme, is an integrated extension to it, and cooperates with it.

The current scheme fits within the architecture of "Ring Zero Demultiplexing", which is a scheme put forth in Release 7.0 whereby a hierarchy of software levels called "multiplexers" are interposed between the user's physical terminal and the common code of MCS. For example, the code which manages the communications processor is a multiplexer (known as "fnp_multiplexer"), which manages most TTY channels. Ring Zero MCS does not call fnp_multiplexer directly, but rather via a standard set of interfaces (channel_manager), specifying the device-index of the terminal under consideration. Channel_manager realizes that fnp_multiplexer is responsible for that terminal, and dispatches to the latter. Similarly, when fnp_multiplexer processes an interrupt from the communications processor, it determines which channel is involved, and calls the channel manager interrupt entry to call the interrupt side of ring zero MCS (tty_interrupt) to do "TTY interrupt-side processing", specifying one of a standard repertoire of "interrupt types" along (perhaps) with data received from the front-end. Fnp_multiplexer may be thought of as a procedure multiplexing and de-multiplexing the single interface to the communications processor.

The point of this "Multiplexer" protocol is that a device such as a terminal concentrator may be attached to Multics via one channel of the Front-End (or even some other path into Multics), and have its demultiplexing/routing mechanism managed automatically as part of this

scheme. If such a terminal concentrator exists, requests made by ring-zero MCS to control (e.g., output to) terminals on it will be forwarded by channel_manager to the concentrator-manager procedure, which will perform the necessary encodings as required by the concentrator's protocol, and call channel_manager recursively to pass the new data on to fnp_multiplexer to control the actual channel. Similarly, interrupts received by fnp_multiplexer for the concentrator's channel will be forwarded by channel_manager not to tty_interrupt, but to the concentrator-manager procedure. The latter will interpret the received data, and call channel_manager to invoke tty_interrupt for the appropriate terminal.

It is via this means that arbitrarily-nested multiplexed communications channels may be supported, as long as the multiplexing/demultiplexing procedure exists within the supervisor. The set of calls from Ring Zero MCS, or lower-level multiplexers, to each "multiplexer" consists of calls to read and write data, and peform a well-defined set of "control orders" upon the channel. Each multiplexer has the right to perform or refuse to recognize any of these control orders, indicating which via a status code. The set of calls from the interrupt side of the multiplexer back to superior multiplexers, or the interrupt side of Ring-Zero MCS, consists of the set of "interrupts", some with data, some without.

It is within this architecture that the FNP Echo negotiation (or more accurately, Echo negotiation by multiplexers) is designed. It consists of a new repertoire of control orders to multiplexers and interrupts from multiplexers, via which echoing by the multiplexer is managed in an organized way.

In this scheme, a multiplexer either "knows how" to perform negotiated echo, or does not. Supporting negotiated echo means possessing the ability to accept a break-character table, and echo characters according to it, until one of the "echo negotiation break conditions" described in the previous paper are encountered, in response to the new control orders. The FNP (via fnp_multiplexer) knows how to echo (which we will take to mean "negotiated echo via this protocol"). This specifically does not mean that multiplexed channels that happen to go through the FNP "inherit" this ability: they do not. For channels whose multiplexers cannot support this protocol, ring zero MCS is prepared to perform the echoing itself as it does today. As it turns out, ring zero must be prepared to perform this echoing even for multiplexers that do support the protocol.

The multiplexer which supports negotiated echo will, when so requested, echo characters as they arrive from the terminal and not hand them to MCS until one of the echo negotiation break conditions is recognized by it. Inception of echoing by the multiplexer is started by a call from MCS, when MCS is told to begin echoing, and other conditions are ripe. Input characters are handed by multiplexers to MCS via the "ACCEPT_INPUT" interrupt ("interrupt" as defined above in the multiplexer/MCS layering). In addition to the input characters,

the ACCEPT_INPUT interrupt conveys (today) a bit, "break_character" which states whether or not a "break character" (normally CR, LF, etc.) was placed in the input by the multiplexer. In "breakall mode" (character-at-a-time processing), every such character delivery today has this bit "on". Under the new protocol, for a multiplexer which supports negotiated echo, this bit is used to indicate whether the multiplexer itself echoed the characters in this delivery. If the bit is on, the delivery consists of characters none of which were echoed by the multiplexer; the multiplexer may decide for any reason (e.g., internal buffer shortages, internal races, etc.) to stop echoing (as can ring zero vis-a-vis ring 4). Of course, it must stop echoing for the defined echo negotiation break conditions. If the "break_character" bit is off, the delivery consists of characters all of which were echoed by the multiplexer, except for perhaps the last character of the delivery. MCS must determine, for such a delivery, whether the last character of such a delivery was capable of being echoed by the multiplexer, and if so, assume that it was, otherwise not. This "wart" makes it unnecessary to make two distinct deliveries (to distinguish the echoed from non-echoed characters) every time a break condition is encountered.

When Ring Zero MCS receives an echo-break table from ring 4, it will attempt to ship the break table to the Multiplexer via a new control order ("set_echnego_break_table"). By virtue of the way negotiated echo (in general) works, MCS (Ring Zero or otherwise) cannot be in the echoing state when this table is set. Thus, both Ring Zero and the multiplexer (e.g., the FNP) will have an appropriate (and identical) break table when the the call to begin negotiated echo is made.

When Ring Zero MCS is called upon to begin negotiated echo (echo_negotiate_get_chars, as in the previous paper), and has no characters to deliver immediately (already accumulated), it calls upon the inferior multiplexer to "start negotiated echo", via a control order, also specifying the number of characters left on the line, as in the protocol of the previous paper. If this control order is refused (the multiplexer does not support the new protocol (of course, fnp_multiplexer in specific, does)), ring zero proceeds as today, with interrupt-side negotiated echo. If the multiplexer goes along with the order, the channel is marked (in the ring-zero echo data) as having a multiplexer knowledgeable about the protocol. In either case, ring zero will enter the "ring zero echo state" (of the previous paper).

After this operation has been performed, the multiplexer (e.g., the FNP) will either be echoing characters or not: by virtue of the synchronization protocol (to be described), the multiplexer will either have honored the request or not. MCS will not know whether the request has been honored or not until characters arrive from the multiplexer at the MCS interrupt side.

If the multiplexer delivers up a shipment of characters with the "break_character" bit on, either the request was not honored, or the specific multiplexer does not support echo negotiation, the multiplexer decided randomly (i.e., for internal reasons) to stop or not start

echoing, or the first character in the delivery is a break character or exceeds the length of screen left: in any case, the delivery is entirely of non-echoed characters. These cases are indistinguishable from each other and from the only case today when in the ring zero echo state, and handled identically as today. The delivery is scanned, a possible leading prefix of echoable characters echoed by ring zero, and the characters made available (if ring zero leaves the ring zero echo state while processing them) to ring 4, which would then be woken up.

If the multiplexer delivers up a non-empty shipment of characters without the break_character bit on, and the specific multiplexer has been found to be knowledgeable about multiplexer echo negotiation (i.e., earlier accepted the "start negotiated echo" control order) all characters except the last in the shipment are known to have been echoed by the multiplexer, which has apparently honored the order call which was issued at the time ring zero went into the echoing state. Thus, this only possible if ring zero is in the echoing state. Those characters are counted by ring zero as "having been echoed by ring zero" (as far is ring four is concerned) and are not echoed by ring zero. The last character is checked for stopping ring-zero echo, and if it would not stop ring zero echo, is treated as one of the multiplexer-echoed characters. If it would stop ring zero echo, it is processed as today, indeed takes ring zero out of the echo state, and causes ring 4 to be woken up, as today.

Whenever the multiplexer delivers to the Ring Zero MCS interrupt side a character that takes ring zero out of the echo state, the multiplexer itself will be known to have stopped echoing. Thus, ring zero leaves the echoing state after the multiplexer stops echoing. Note also that ring zero enters the echoing state upon receipt of the echo_negotiate_get_chars call, which is before the multiplexer enters its echoing state (upon processing the receipt of the control order sent at the time ring zero enters the echoing state. Thus, the multiplexer's echoing-state interval is completely and properly contained within that of ring zero.

There is an implicit race condition associated with the start of Multiplexer echoing: when ring zero enters the echoing state and sends the order to the multiplexer to begin echoing characters, ring zero (or ring 4, for that matter) does not know whether or not the multiplexer has processed some input characters, and sent them on their way, which have not yet arrived in ring zero. This is not unlike the case today where characters have been received by the multiplexer and not yet handed to ring zero when ring zero enters the echo state. Today, these characters will eventually be processed (they will be the next characters to be processed) and no harm is done. Yet, when the multiplexer has the ability to echo, the order to start echoing may be invalid when received by the multiplexer if the latter has processed characters not yet seen by ring zero. If these characters are echoable, the ring zero interrupt side would attempt to echo them (which would be in the wrong order with echoing being done by the multiplexer). If they are not echoable, the order for the multiplexer to start echoing is in error, for the unprocessed characters may

produce output (when processed by ring 4) which must precede any echoing done by the multiplexer (e.g., cursor positioning, buffer switching, or any arbitrarily complicated request with arbitrarily complicated effects on the screen.

Thus, the multiplexer must not start echoing (i.e., not honor the order to start echoing) if it has processed (but not had received on the ring zero interrupt side) characters at the time the start_negotiated_echo order is received at the point which processes characters. This determination is made by the "input processor" of the multiplexer based upon a value called the synchronization counter sent with the start_negotiated_echo control order: the value sent by ring zero is the count of all characters received by the ring zero interrupt side since the last character echoed by the multiplexer. The multiplexer input processor will also count characters processed by it since it last echoed a character. Since the start_negotiated_echo control order can only be received when the multiplexer is not in the echoing state, any multiplexer-but-not-ring-zero processed characters could not have been echoed by the multiplexer. Thus, the equality of the multiplexer input processor's count and the count received from ring zero in the start_negotiated_echo control order is necessary and sufficient reason for the multiplexer to enter the echoing state, and begin echoing characters. If the counts are not equal, it must be the case that non-echoed characters are "in transit", and the order must not be honored. Ring zero's shipped count can only be less than the multiplexer's, or equal: it can never be greater.

The above protocol has the effect of multiplexers starting echo only when there are no characters "in transit" between the input processor and the ring zero interrupt side. This would have the effect of each ring 4 echo_negotiate_get_chars being entirely echoed by ring zero or the multiplexer, as there be characters in transit or not at the time of the receipt of the start_negotiated_echo control order by the multiplexer's input processor. We would like the multiplexer to perform as much echoing as possible, even if the start of this echo_negotiate_get_chars lost the race condition. Thus, when ring zero processes (and echoes) an entire delivery of echoable characters from a multiplexer which claims to know how to perform negotiated echo, it will (after having so done) send out another start_negotiated_echo control order, which again, may or may not succeed when it reaches the input processor.

There is a fairly subtle problem here when long communication lines or packet-switched networks present a time delay between the Multics Central System and the echo-negotiating processor that is longer than the inter-character typing time. If the typist continues to type faster than the communication line/system can send characters and receive the control orders, there will always be characters "in the pipe", and not only will all the start_negotiated_echo control orders fail, but more interrupts/communications than today will occur due to the extra control order at every such character! Thus, until the typist scratches his or her head, etc., the system can never synchronize, and will generate extra overhead! With the FNP doing echo-negotation, this should not be a problem, as the round-trip

response time of the network from the FNP through ring zero and out again to the FNP is substantially less than an input character-time. (Note that line speed is completely irrelevant: only the inter-character time of the typist is an issue).

Another thorny point of the present scheme is the initialization of the synchronization counters: If they count characters processed since the last multiplexer-echoed character, what value are they to have if the multiplexer has never echoed any characters? It will never be able to start echoing, since successful inception of echoing depends upon the values of the synchronization counters. This is solved by another new control order, "init_echo_negotiation", which must be supported by all multiplexers supporting echo negotiation. Ring Zero MCS will maintain, for each channel which has ever dealt with echo negotiation, a "synchronized" flag, initially off. As long as this flag is off, received characters are not counted against the synchronization counter. When a call to echo_negotiate_get_chars is made upon ring zero, and there are no undelivered (to ring 4) characters in ring zero, the "synchronized" flag will be inspected. If on, action will proceed as above. If off, new control order will be issued to the multiplexer. If not honored, ring zero enters the echoing state as usual, knowing that the multiplexer will not ever echo. If honored, the echoing state is not entered. Characters will arrive, non-echoed, from the multiplexer, and cause ring 4 wakeups. At the time the multiplexer's input processor (which maintains the multiplexer's synchronization counter) receives the init_echo_negotiation control order, it zeroes its synchronization counter, begins counting characters (it must be in the non-echoing state) thereafter, and sends a new type of interrupt to Ring Zero MCS, ACK_ECHNEGO_START. When ring zero receives this interrupt, it zeroes its synchronization counter, turns on the "synchronized" bit, and begins counting received non-echoed characters. Characters are currently not being echoed: they are being delivered to ring 4 unechoed as they arrive. The first character received in ring 4 will cause ring 4 to promptly make another echo_negotiate_get_chars call (as ring 4 is still in its echoing state), and all proceeds as above, when synchronized.

One more service needed of MCS echo negotiation is the ability to stop MCS echoing on demand of ring 4, when the latter receives some non-keyboard-initiated event (e.g., a console message) which will affect the display. When such an event occurs, ring 4 will call into ring zero to stop echoing, and make calls to determine precisely what has been echoed, i.e., what is on the screen, with the certain knowledge that MCS will echo no more until instructed to do so.
Today, this is done (idea due to Spencer Love) by an echo_negotiate_get_chars call with zero "screen length left". Ring zero, when so called, will return what it has received, indicate how much it has echoed, and stop echoing in a unitary operation. The zero screen length left will prevent it from re-entering the echoing state (or staying in it), which it would otherwise do. The complex scheme described in r0-inter.text (BSGR0I >) was never implemented, and was discarded in light of this superior idea.

When multiplexer echoing is involved, not only ring zero, but the multiplexer, must leave the echoing state when such a call is made. Although ring zero could leave the echoing state instantly when such a call is received, ring 4 cannot proceed knowing that echoing has stopped until there is positive verification that the multiplexer has stopped echoing. Thus, the following protocol has been devised: At the time of an asynchronous ring 4 event (Emacs interrupt), Ring 4 will be prepared to try and retry the call to call hcs_$echo_negotiate_get_chars with a screen-length-left in a loop, until the entry indicates that echoing has indeed stopped. The entry will return a zero code and actually return characters if echoing has stopped, or return error_table_$line_status_pending if it has not, in which case ring 4 is expected to block. In this way will ring 4 wait for the multiplexer to acknowledge stopping echoing. Upon receipt of such a call, ring zero will, if in the echoing state, and has not already done so, send out a new control order to the multiplexer, "stop_negotiated_echo", and set a bit saying that it has done so. If called in this way when in the echoing state, and the order call is supported, ring zero will return error_table_$line_status_pending. This code will be returned for each such call until the echoing state is exited as is to be described. When the multiplexer input processor (which does the actual echoing) receives this control order, it will stop echoing (leave its echoing state), having echoed and forwarded all it is going to echo) if in its echoing state, and send another new type of interrupt, ACK_ECHNEGO_STOP, to the MCS interrupt side. The MCS interrupt side will receive this <u>after</u> it has processed all characters processed by the multiplexer input processor before the latter received the control order, and thus, ring zero knows at this time exactly how much was echoed and that no more will be echoed. At this time, the MCS interrupt side takes ring zero out of the echoing state, turns of the bit saying that the control order was issued, and wakes up ring 4. Ring 4 wakes up, retries the call, receives a zero error code, and real characters (including a count of how many were echoed), and echoing does not restart, because of the zero screen-left passed in as input on the last (indeed each) of these calls.

One further point needs be mentioned. A Multiplexer may find that it cannot honor a start_negotiated_echo control order because it has output which it has already accepted from Ring Zero MCS queued internally. fnp_multiplexer, for instance, cannot send the control order to the FNP until the internally queued output is taken by the FNP, lest echoed output, echoed by the FNP, appear in the wrong sequence with respect to Multics Central System output. The multiplexer will, in this case, return the code error_table_$invalid_write to Ring Zero MCS, which will set a special flag bit indicating that when the multiplexer asks for more output (via the SEND_OUTPUT interrupt), another attempt is to be made to send the control order to the multiplexer before any more output is given to it. Such a "failure" is not considered to be a case of the multiplexer rejecting (or not honoring) the control order. This scenario will almost certainly occur in the case mentioned above where a second attempt at synchronization for a given echo negotiation causes a start_negotatiated_echo control order to be sent after ring zero has echoed characters following a muliplexer synchronization failure.

(END)

(END)